ARES v1.2

The Necessary Theory Part 1/3

The first three chapters aim to cover all the necessary theory as well as the concepts on which the practical part of this course is based. We will start with a short description about what Reverse Engineering is and the reasons why someone might need it, and then we'll proceed with more technical concepts. During the first three chapters we will be discussing the basics behind the Intel IA-32 CPU architecture (x86), the stack, the heaps, as well as the exceptions, Windows APIs with some Windows Internals, and the most common types of reversing tools used these days.

Table of Contents

- Introduction
- What is Reverse Engineering
- Do we need Reverse Engineering?
- The basics behind the Intel IA-32 CPU architecture
    - General Purpose Registers ([1] - Vol 1, Chapter 3.4.1)
    - EFLAGS Register ([1] - Vol 1, Chapter 3.4.3)
    - Segment Registers ([1] - Vol 1, Chapter 3.4.2)
    - Instruction Pointer Register ([1] - Vol 1, Chapter 3.5)
    - Debug Registers ([1] - Vol 3, Chapter 317.2)
    - Machine Specific Registers (MSRs) ([1] - Vol 1, Chapter 3.2)
- Conclusions
- References

Introduction

The first three chapters aim to cover all the necessary theory as well as the concepts on which the practical part of this course is based. We will start with a short description about what Reverse Engineering is and the reasons why someone might need it, and then we will proceed with more technical concepts.

During the first three chapters we will be discussing the basics behind the Intel IA-32 CPU architecture (x86), the stack, the heaps, as well as exceptions. Windows APIs with some Windows Internals, and the most common type of reversing tools used today.

You will also notice that during these chapters we will be also referring to various x86 assembly instructions. Keep in mind that this is not an assembly programming course, so not much detail will be devoted to this subject, but you can use the bibliography references in order to dig into them in more depth. ([1] - Vol 2).

Probably, you've heard that if you want to become a reverse engineer you have to master assembly language first, which is far from the truth. Of course, being a programmer, especially in a not very high level language like C/C++ can be very helpful in some concepts like pointers, etc., but in regards to reversing and assembly language, what you really need to be able to understand what you see. There's no need to write complex programs using assembly.

In fact, the main goal of the reversing process is to truly understand what you are analyzing and to be able to reproduce your findings in any programming language you choose.

What is Reverse Engineering

Reverse engineering is the understanding of the internals of something made by a human, through analysis, without having access to its design principles and the way its components interact them in order to make it work.

In other words, it's the process of taking apart something that someone else built and understanding how he did it, partially or completely, so that you are able to make something on your own that can achieve the same purpose. In this course we will refer only to software, but the same concept applies to hardware.

Do we need Reverse Engineering?

Imagine that you own third-party software which is not supported anymore by the company or the individual that developed it, and has a tiny but very annoying bug: something that you think you could easily fix, but obviously - you don't have access to the source code.

Or, imagine that you have a passion for malware analysis. You won't have access ot the source code, but you still want to know how the malware works, or even better, you want to be able to analyze that tiny piece of software that is in your hands. You want to know how it infects a computer, what tricks it uses in order to evade anti-virus.

In this case, you are a software developer that doesn't have the budget to buy expensive third-party commercial software protection. Are you going to leave your software unprotected? Of course not!

Understanding the concepts behind software reverse engineering can help you in many cases. You might think you will never need it, especially if it's not your every day job in the IT industry, but that one time you need it, it might save you from a lot of trouble. Of course, there is always the chance that you have a lot of fun doing it, so why not give it a shot?

The basics behind the Intel IA-32 CPU architecture

We are mostly interested in registers, which are small units of internal memory that live inside the CPU.

The fact that these small units of memory reside inside the CPU gives the advantage of very fast access to the data stored inside them. This makes them really useful for executing instructions such as addition, subtraction, multiplication, and division between integers, as well as for performing logical operations such as OR, AND, XOR, NOT, etc.

During this chapter we'll take a look at those registers that we are most interesting in while reverse engineering applications.

General Purpose Registers ([1] - Vol 1, Chapter 3.4.1)

In fact, the fastest way to access data is to store it inside a register. However their capacity is very limited and, in the Intel CPU architecture we're talking about, there are only eight of them that we can really use for generic purposes.

These registers are also called general purpose registers and their capacity is 32 bits: 4 bytes (4 sets of 8 bits):

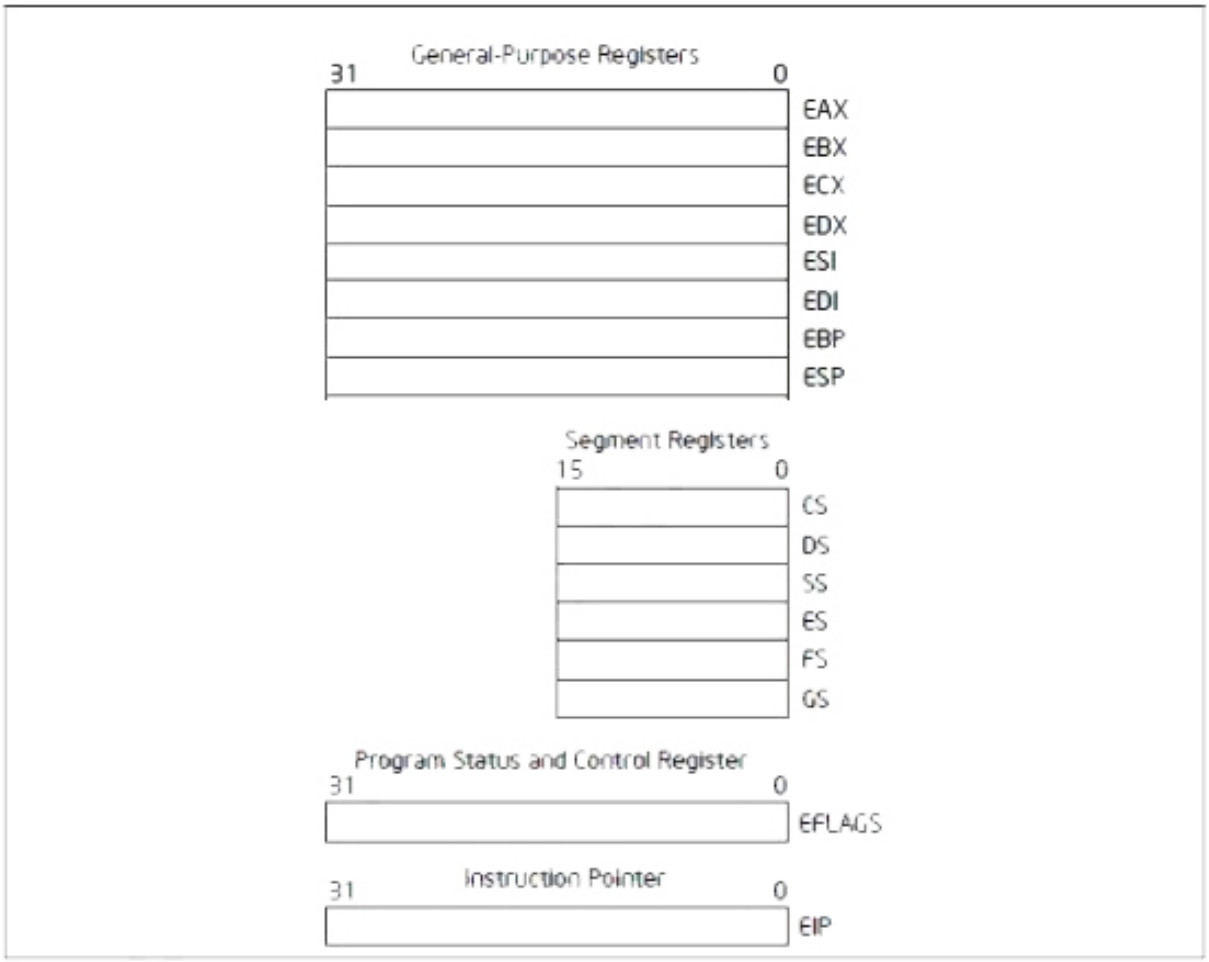

Figure 1.1 Intel IA-32 General Purpose Registers

The following figure summarizes some special uses these registers have:

- EAX; Accumulator of operands and result data
- EBX; Pointer to data in the DS segment
- ECX; Counter for string and loop operations
- EDX; I/O pointer
- ESI; Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- EDI; Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- ESP; Stack pointer (in the SS segment)
- EBP; Pointer to data on the stack (in the SS segment)

It's also possible to have to access specific portions of their memory, as the following figure demonstrates:

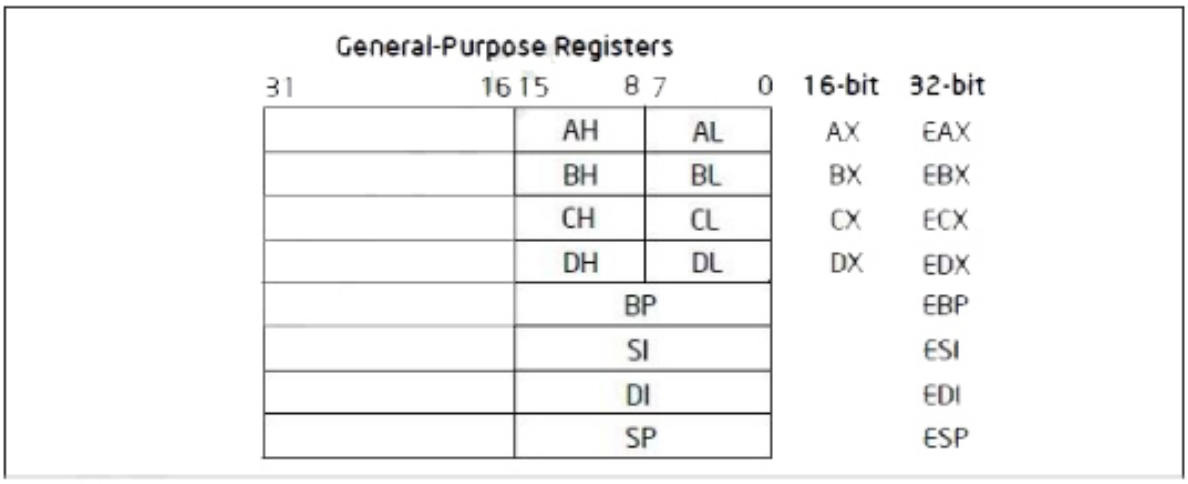

Figure 1.3 Intel IA-32 Chunks of General Purpose Registers

As seen in the figure above, for the EAX, EBX, ECX, and EDX registers, it's possible to have access to the individual 8-bit parts of the low 16-bit chunk.

In other words, we can access only the high byte of AX, called AH, or just the low byte of AX, called AL, if we want.

For the rest of the general purpose registers mentioned above, we can only access their low 16-bit memory chunk as a whole.

Example:

AX = 3256h

mov al, 08h ← access low byte of AX → AX = 3208h

mov ah, 45h ← access high byte of AX → AX = 4508h

BUT:

DI = 4856h

Mov DI, 0008 ← access the whole 16-bit chunk of EDI → DI = 0008h

Code Snippet 1.1 Accessing lower memory chunks of General Purpose Registers

EFLAGS Register ([1] - Vol 1, Chapter 3.4.3)

Another very important CPU register that we need to mention at this point is the EFLAGS register. This is a collection of 1-bit flags, shown below.

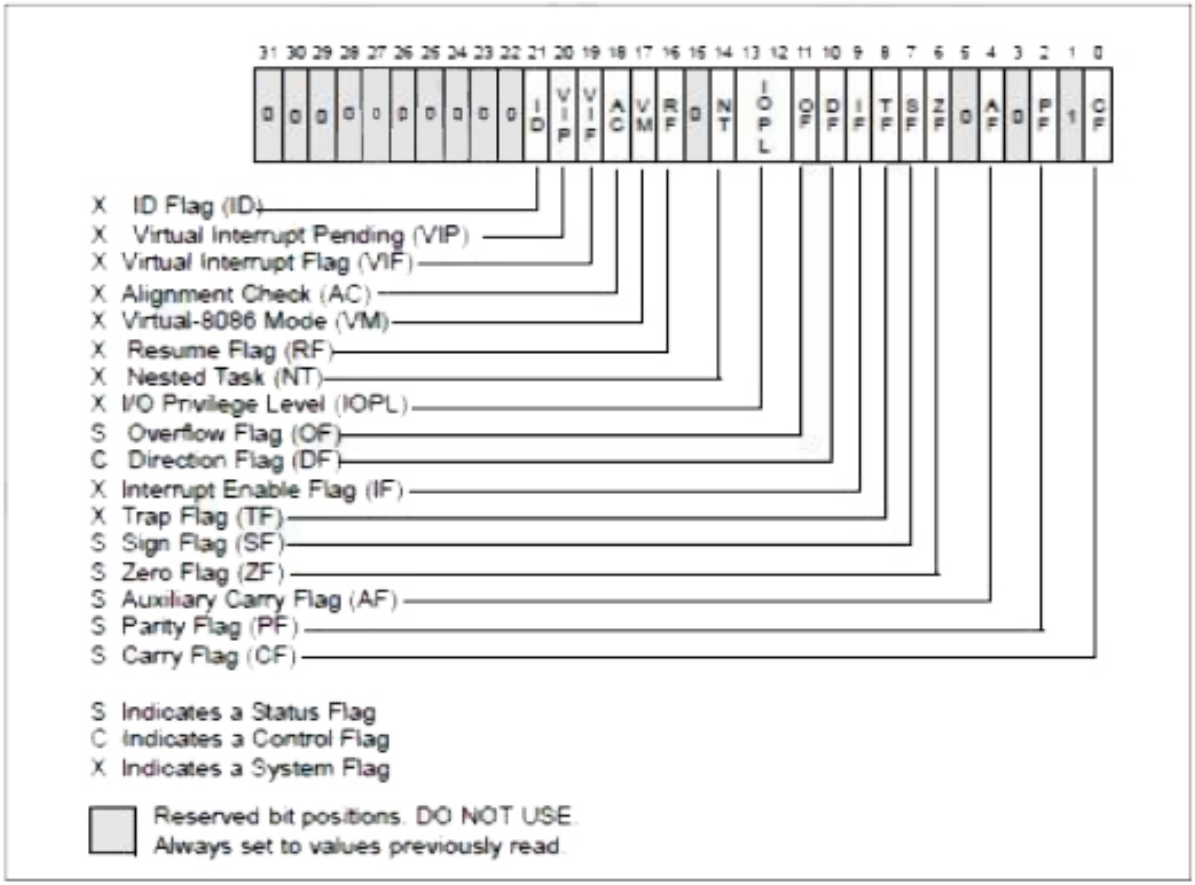
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE
Always set to values previously read

Figure 1.4 EFLAGS Register

In this register is information related to the status flags, the control flag, and the system flags.

The status flags (see Figure 4) are directly related to specific types of instructions. The arithmetic instructions make use of the OF, SF, ZF, AF, PF, and CF flags.

On the other hand, the SCAS (Scan String), CMPS (Compare String - cmsb, cmpsw, cmpsd), and LOOP (LOOPE, LOOPZ, LOOPNE, LOOPNZ) instructions make use of the ZF flag in order to indicate the completion of their operations (and, in some cases, the result.)

For example, if we use the 'repe cmpsb' instruction to compare two strings, byte by byte, and the two strings are equal, the ZF flag will be set to 1. Otherwise, it will be set to 0.

The control flag DF is used to control instructions related to string processing. For example, if DF is set (meaning DF=1), the string instructions auto-decrement, so that they are processed from higher to lower addresses. On the other hand, if DF is not set (DF=0), the string instructions auto-increment,, so they will process the strings form lower to higher addresses.

Usually, we set DF to 0, by using the 'cld' instruction (clear direction flag), before processing strings, in order to process them from the beginning to the end,, as we would read them. Sometimes, though, the 'std' instruction 9set direction flag) is used to set the DF to 1 in order to process the strings from the end to the beginning.

it is important to mention that the ESI and EDI registers that rely on these flags must point either to the start or the end of the strings before starting this operation. At the same time, the ECX register must contain the number of bytes we wish to compare, for example: the length of the two strings (if they both have the same length.)

The system flags and the IOPL (I/O privilege level) field inside the EFLAGS register are involved with operating system operations.

We are not terribly interested in the system flags, except for one: the Trap flag, which enables single-step mode. This generates a single-step exception after the execution of each instruction, and is critical for debugging purposes in order for us to keep control of the execution of every single instruction of a program under inspection.

Segment Registers ([1] - Vol 1, Chapter 3.4.2)

In addition to these common registers and flags, we also have a group of 16-bit registers called segment registers which contain special pointers called segment selectors that identify the different types of segments in memory. In order to access a particular segment in memory, the appropriate segment register must contain the correct segment selector.

The following figure demonstrates the segment registers according to the flat memory model used by Windows NT OS, which is structured so that applications see the available physical memory as an array of memory locations. The OS takes care of the rest, such as denying access to applications into the kernel of the operating system, protecting applications from each other so that no application can unintentionally interfere with another, etc [2]

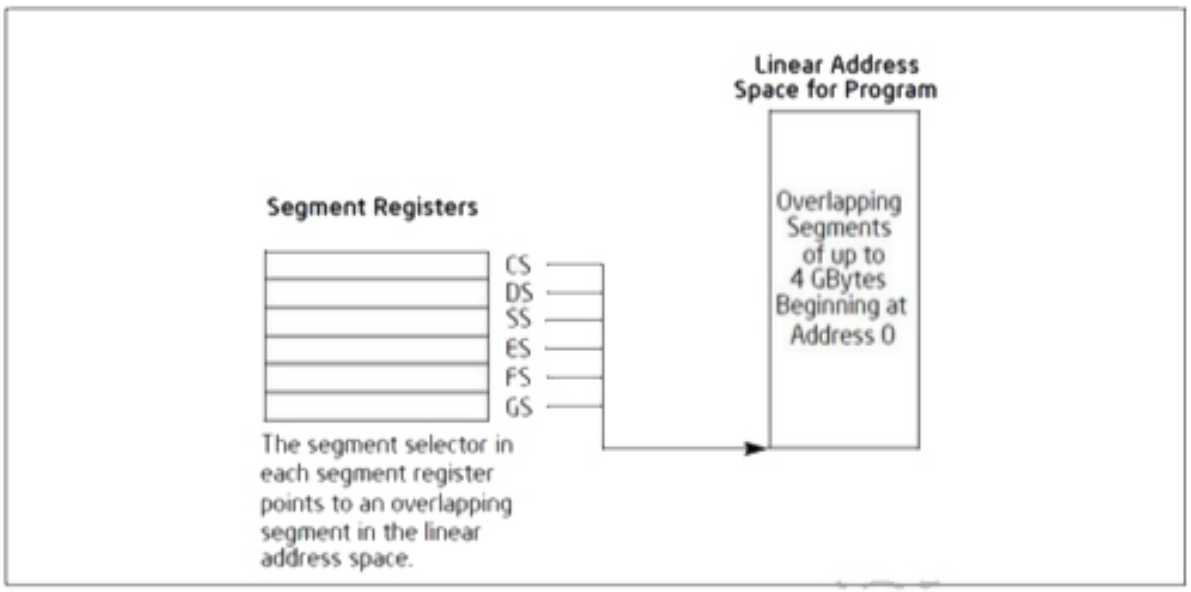

Figure 1.5 Segment Registers

Each one of the segment registers points to a specific type of storage: code, data, or the stack. More information about the stack is coming in Chapter 2.

As you can see in Figure 1.5, the CS register contains the segment selector for the code segment, which is the memory area that stores the instructions that are being executed.

DS, ES, FS, and GS registers point to four different data segments, which are used to store different types of data structures or single variables.

Finally, the SS register points to the stack segment, where the stack of the current thread under execution is stored in memory. For this reason, all stack-related operations use the SS register to locate the stack segment.

Instruction Pointer Register ([1] - Vol 1, Chapter 3.5)

We also have the instruction pointer register (EIP), also called the program counter (PC), which points to the next instruction to be executed in the code segment. Every time an instruction is executed, the EIP is updated to point at the next instruction.

Since the EIP register cannot be accessed directly, in the case that we may need to read the address of the next instruction to be executed, we'll need to pull a simple trick that will be described later in Chapter 2, 2.6, since it involves stack-related operations.

Debug Registers ([1] - Vol 3, Chapter 317.2)

These debug registers, as their name implies, are used to control the debug operation of the processor. There are eight of them labeled as DR0-DR7 (DR0, DR1, DR2... etc.)

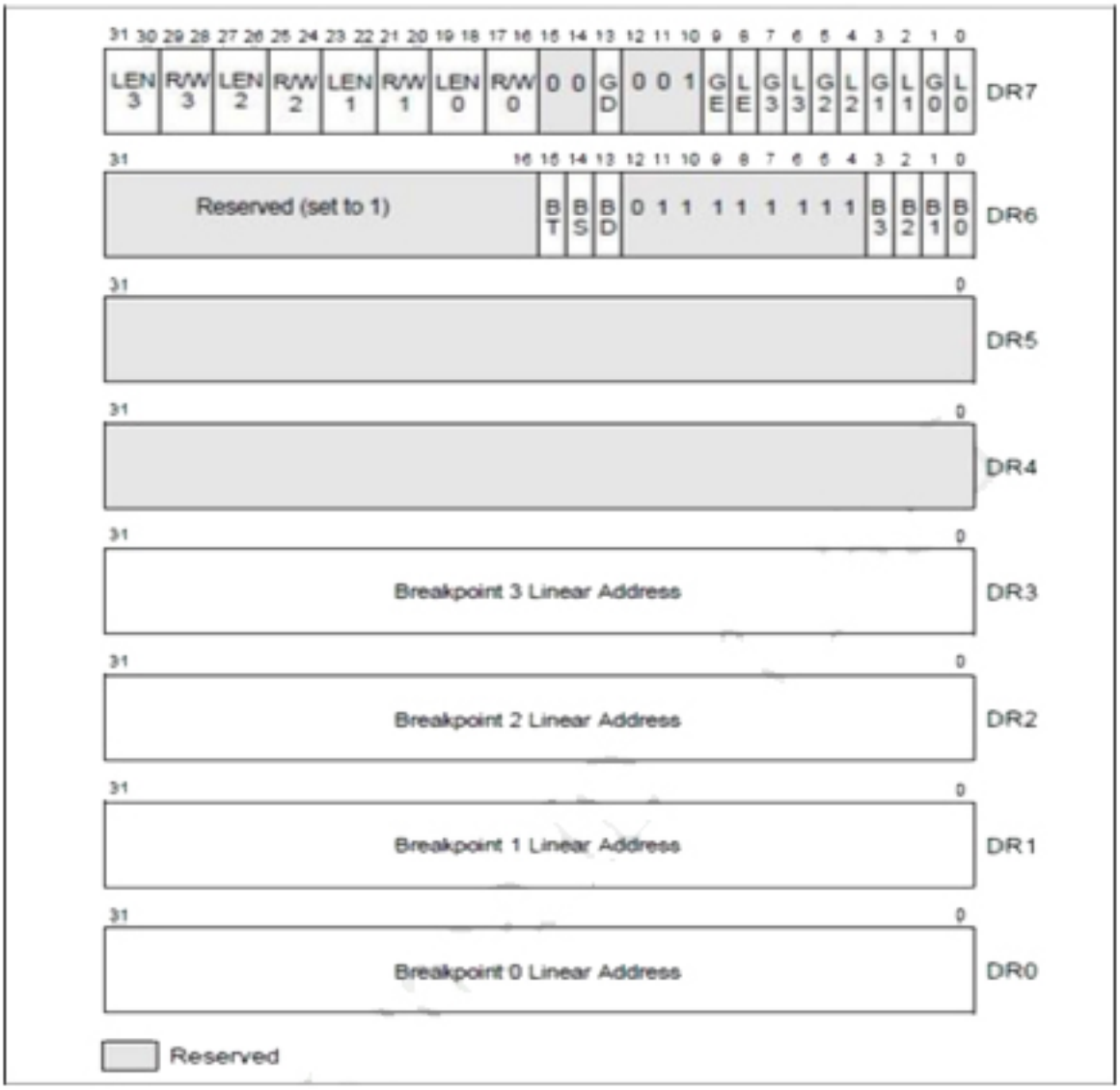

Figure 1.6 Debug Registers

In the context of reverse engineering, we are mostly interested in the first four debug registers DR0-DR3, which are used to store hardware breakpoints on specific addresses which will be triggered if a desired condition is met.

In other words, DR0-Dr3 registers contain addresses where, if a specific type of memory access occurs, the execution of the program will pause, giving us the opportunity to examine it under the debugger.

For example, we can set a hardware breakpoint on memory access on a specified address inside the address space of the examined process (see Note below.) We can set this to be triggered if that address is referenced for read or write by an instruction.

Likewise, we can set a hardware breakpoint on execution on a specified memory address where executable code is placed, which will be triggered on every attempt to execute instructions starting from that address.

However, keep in mind that debug registers are privileged resources, which means that we cannot directly access them from the Ring 3 level (also called 'userland') where software is normally executed.

In order to set hardware breakpoints in Windows operating system, we need to use a specific Ring 3 API which will transfer the execution to kernel level in order to update the debug registers. More on this will be discussed later!

Note: Each thread inside a process has its own CPU context which describes the state of the CPU during the execution of that thread. This means that if we set a HW breakpoint on a specific memory area inside the process we really only modify the CPU context related to the debug registers for that thread.

For example, assuming there are no HW bps (breakpoints) enabled at this point, we se a HW breakpoint on access while we debug a specific thread. The address will be stored inside DR0.

Then the process starts a new thread that accesses the same memory area. At this point no HW breakpoints will be triggered since the new thread has its own CPU context, and no HW breakpoints have been set for this specific thread yet.

However, if the debugger is capable of debugging multi-thread applications then it can update the thread context for every thread accordingly in order to enable that HW breakpoint for all the active threads of the process under debug.

Note: Software breakpoints work by substituting the original byte located in the address, where we set the breakpoint, with a 0xCC byte (INT 3h). Since this implies modifying the code in memory, then these are completely independent from the thread context, thus always effective regardless of whether the process is multi-threaded.

Machine Specific Registers (MSRs) ([1] - Vol 1, Chapter 3.2)

These registers are also called "Model-Specific Registers". They handle system-related functions and they are not accessible to applications, except from the Time-Stamp Counter.

This is a 64-bit register, and its content can be read using the RDTSC instruction, which stands for Read Time-Stamp Counter.

The low-order 32 bits are loaded into the EAX registers, while the high order 32 bits are loaded into the EDX register.

The Time-Stamp Counter register is increased by the processor at every clock cycle. it will reset to zero when the processor is reset.

More information on the reasons why we might be particularly interested in this register and the RDTSC instruction will be discussed in Chapter 11.

Conclusions

Now that you have a decent overview about the important-to-us processor registers, we should proceed with other important information and concepts that we'll need ruing this reverse engineering course.

Feel free to re-read this at any time. An understanding of registers, what they do, and how they can be read and written is fundamental to any reverse-engineering exercise, so make sure that you are at least comfortable with the idea, and we'll show you more detail later.

References

1. Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C.
2. Lesson 7 - The Virtual Memory Manager http://technet.microsoft.com/en-us/library/cc767886.aspx